LOGiiX™

User's Manual for GUIDE® Author™

InfoAccess™ INCORPORATED

LOGiiX™ User's Manual

All *GUIDE*® documentation and training materials are copyrighted, and all rights are reserved. Except as authorized in the terms of a valid license agreement, neither the documentation nor any software that accompanies it may be reproduced, translated, or reduced to any electronic or printed form without the prior consent of InfoAccess™ Inc.

Copyright © 1998 InfoAccess Inc. All Rights Reserved.
Printed March 1998 in the United States.

InfoAccess, the InfoAccess logo, Table Viewer DLL, *GUIDE* Table Viewer Style Editor, Style Markup Format (SMF), and Table Markup Format (TMF) are trademarks of InfoAccess Inc.

GUIDE is a registered trademark and *GUIDE Author, GUIDE Indexer, GUIDE Professional Publisher, GUIDE Reader, GUIDE Viewer, GUIDE Writer, GUIDE Writer Style Editor, LOGiiX,* and Hypertext Markup Language (HML) are trademarks of Office Workstations Limited licensed to InfoAccess Inc.

Other trademarks and registered trademarks are the property of their respective owners.

Information is subject to change without notice.

InfoAccess Inc.

15821 NE 8th St
Bellevue, WA 98008-3905
USA

Technical Support

Phone 425-201-1916
Email techsupp@infoaccess.com

Corporate Headquarters

Phone 425-201-1915
Sales 800-344-9737
Fax 425-201-1922
Web www.infoaccess.com
Email info@infoaccess.com

MAN5000-03B

Contents

1 WELCOME TO LOGIIX

About This Manual 6
Getting Started 7

2 LOGIIX FUNDAMENTALS

LOGiiX Scripts 9
Blocks 10
Data Types 12
Constants 13
Predefined Constants 13
Variables 14
Identifiers 16
Variable Scope 17
Global Variables 18
Reserved Words 19
Limitations 20

3 LOGIIX OPERATIONS AND LANGUAGE STATEMENTS

Expressions 23
String Operators 23
Statements 25
- Assignment Statements 25
- If Statements 26
- While Statements 27
- Include Statements 27
- Return Statements 29

Comments 30
Functions 30
Special LOGiiX Definitions 32
Dynamic Data Exchange (DDE) 33
- LOGiiX as a DDE Server 33
- Initiating a DDE Link 34
- Using DDE Execute 36
- LOGiiX as a DDE Client 37
- DDE Functions Example 38

4 USING LOGIIX WITH MULTIMEDIA

Device Types 40
Using LOGiiX Multimedia Functions 41
Using mciSendString 41
Using mciWaitNotify 42
Using mciGetLastError 43
Using mciGetErrorString 43
Using Audio 44
Using Video 45
An Advanced Technique 45
Troubleshooting 47

5 LOGIIX ERROR MESSAGES 49

INDEX 55

CHAPTER 1

WELCOME TO LOGIIX

LOGiiX is a high-level macro scripting language designed for use with *GUIDE* electronic publishing tools. You can use *LOGiiX* to write scripts that open and close documents, display messages on the screen, and activate commands even when the menu bar is not visible. As you learn more about *LOGiiX*, you can create more complex scripts that automatically create hyperlinks, modify *GUIDE* document contents, keep track of who uses your documents, and communicate with other Windows-based applications.

LOGiiX provides all of the facilities of a complete macro scripting language. In addition, it also provides functions to control the *GUIDE* environment. *LOGiiX* features include:

- Arithmetic and logical operations
- Text string functions
- Access to all menu commands and many dialog options in *GUIDE Author* and *GUIDE Reader*.
- *GUIDE* document manipulation
- Looping constructs (**if...then...else**, **while...do**)
- Access to Objects in *GUIDE* documents, including their data and attributes
- File input and output
- Support for Windows' Dynamic Data Exchange (DDE) feature

You can use *LOGiiX* to extend the functionality of *GUIDE* publications to handle the most demanding tasks. *LOGiiX* also enables you to hide complex document manipulation and navigation from readers so that your *GUIDE* electronic publications remain intuitive and easy to use.

LOGiiX syntax is similar to the Pascal programming language. If you are already familiar with Pascal, you'll learn to write *LOGiiX* scripts quickly with minimal help. If you've never used a scripting language before, please read this manual carefully. For detailed information about *LOGiiX* functions, see the *LOGiiX Command Reference,* which is part of *GUIDE Author's* online help system.

A *LOGiiX* script is a list of instructions contained in a Definition linked to a *GUIDE* Command Button. When a reader clicks on the Command Button in a *GUIDE* document, the Command Definition passes these instructions to *LOGiiX*. The script is compiled into an intermediate stack-based machine language, and then interpreted and executed by a pseudo-machine—a 'computer' implemented entirely in software. These intermediate stages remain hidden: when readers click on a Command Button, any linked *LOGiiX* script is compiled, interpreted, and executed in a single step.

About This Manual

This manual contains:

- An introduction to the *LOGiiX* scripting language
- Basic information about *LOGiiX* scripting including explanations of reserved words, variables, and their use in scripts
- Information about operators, statements, blocks, functions, limitations, and error messages in *LOGiiX*

This manual provides all the information you need to incorporate *LOGiiX* scripts into *GUIDE* documents, whether you've never written a macro script or you're experienced with other programming languages. Since *LOGiiX* is used to write Definitions for Command Buttons in *GUIDE* publications, you should be familiar with *GUIDE* terms and technology before you begin using *LOGiiX*.

For detailed information about *LOGiiX* statements and functions, please see the *LOGiiX Command Reference,* provided as part of *GUIDE Author's* online help system. For more information about commands and operations in *GUIDE Author, GUIDE Reader,* and *GUIDE Viewer,* please refer to *Welcome to GUIDE Author*, the

GUIDE Author User's Manual, the online *GUIDE Author Command Reference,* and the *GUIDE Reader User's Manual.*

An important note about *LOGiiX* scripts shown in this manual: the printed lines on these pages don't necessarily end where the script lines break on the computer screen. The formatting constraints of the printed page won't allow us to show scripts exactly as they appear onscreen. However, you'll recognize continuation lines that belong with the preceding lines because the continuation lines return to the left margin, not the indentation level where the script line began. The first level of indentation shown in these pages represents the left edge of a computer screen. And, even though *LOGiiX* scripts aren't case-sensitive, we've used upper- and lower-case characters in the scripts shown in this manual so they're easier to read.

```
    This is the first line of a script.
    This is the second line of the script, which is much
longer than the first line. It shows how continuation
lines extend out to the left margin while indents show
where new lines begin.
    This is the third line of the script.
```


Getting Started

The *LOGiiX* compiler, **logiix.dll**, is an interpreter that provides added functionality to *GUIDE* electronic publications. It is installed with the other scripts and files on your installation disks. An entry in the initialization file (**infacces.ini**) identifies the *LOGiiX* interpreter to *GUIDE* applications. Each time you open *GUIDE Author, GUIDE Reader* or *GUIDE Viewer,* the application reads the initialization file to determine which external interpreters are available.

The **infacces.ini** file installed with *GUIDE Author* comes ready to use with all external modules, including *LOGiiX*, correctly identified. To access *LOGiiX*, simply start *GUIDE Author.*

CHAPTER 2

LOGIIX FUNDAMENTALS

This chapter introduces basic *LOGiiX* terms and concepts. We'll also discuss different components of *LOGiiX* scripts and how they are used.

LOGiiX Scripts

A *LOGiiX* script is a series of instructions written in *LOGiiX* syntax. The instructions execute sequentially from the first line to the end of the script unless you explicitly write conditional statements or subroutines. The simplest *LOGiiX* script is:

```
#LOGiiX
Function Main ()
Begin
End
```

This script doesn't actually do anything, but it shows the key syntax every *LOGiiX* script must include. The first line of *LOGiiX* script must be the command **#LOGiiX**. This tells the *GUIDE* Command Definition that contains the script to pass the remaining instructions in the Definition to the *LOGiiX* interpreter for execution. The **#LOGiiX** command can be all uppercase, all lowercase, or both upper- and lower-case.

The next line defines the **Main** function. Every *LOGiiX* script must have a function called **Main**, although it doesn't need to immediately follow the **#LOGiiX** command. Parentheses after a function call enclose any parameters for that function; **Main** has no parameters, so the parentheses are empty.

The next two lines of the sample script are the *body* of the function, also referred to as a block. The body of any function must start with the word **Begin** and terminate with the word **End**. That is, each **Begin** must be balanced with an **End**. If you omit **Begin** or **End**, an error message appears.

Other scripts in this manual and the *LOGiiX Command Reference* may not show this basic syntax. But you should assume that these required lines surround the core lines of an example script. A *LOGiiX* script can be of any length (given memory limitations).

Execution of a *LOGiiX* script *always* starts at the **Begin** statement in the **Main** function. It ends either when an error occurs or when execution reaches the **End** statement that denotes the end of the function **Main**. To force a *LOGiiX* script to stop during execution, press the **Esc** key. This mechanism allows *LOGiiX* users to escape from an infinite loop.

Blocks

LOGiiX is a block-structured language. A block is defined as a group of zero or more language statements contained between a **Begin** keyword and an **End** keyword where each statement contained within the block is followed by a semicolon (**;**). nested blocks are also followed by semicolons.

This is an example of a block of a *LOGiiX* script:

```
Begin
   A := 1;
   B := 2;
   C := A + B;
End;
```

LOGiiX script blocks can be nested to any depth, for example:

```
Begin
    A := 1;
        Begin
        B := 2;
        End;
    C := A + B;
End;
```

Note that blocks are followed by semicolons, as are the statements that are contained within these blocks. The main part of the script that starts with **Begin** and finishes with **End** needs no concluding semicolon. A statement that is part of an **if/then** or a **while/do** statement must end with a semicolon. The exception to this is **if/then/else**, which does not require a semicolon.

The only exception to this rule is a function block. The **End** keyword for a function block must *not* be followed by a semicolon. For example:

```
Function Main ()
Begin
    I := 3;
End
```

Blocks allow you to break up your *LOGiiX* scripts into manageable chunks, making your code smaller, more efficient, and easier to read. For example, code that moves a document window can be written as a function block; you can then call the **MoveWindow** function every time you need to move a document window instead of rewriting the same code.

Data Types

LOGiiX can use three types of data: Boolean values (true/false), integers (numeric values), or strings (text characters). Any *LOGiiX* variable may be assigned a value from any one of these types:

DATA TYPE	*POSSIBLE VALUES*
Boolean	True, false
Integer	-2147483647 to 2147483647
String	"Hello", "Application accepted"

Two special cases apply to strings: new line and tab characters. If you want a string to contain a new line, you must place **\n** in the string:

```
My_string := "This is a line\n and this is another"
```

This string prints as:

This is a line
and this is another

Similarly, if you want tab characters to appear in a text string, place a **\t** in the string; for example:

```
My_string := "one \t two \t three \t four";
```

This string prints as:

one two three four

If the backslash character (\) is required, it must be expressed by placing \\ in the string:

```
My_string := "a\\b\\c\\d";
```

This string prints as:

a\b\c\d

Constants

Constants are values of any supported type that occur in a script. These are examples of constants:

1066

True

"bicycle"

In addition to normal decimal values for an integer, you can refer to character values by enclosing them in single quotes and to hexadecimal constants by preceding them with **0x**. Character values are letters. For example:

'a'	Character constant
0x12af	Hexadecimal constant

Predefined Constants

Some useful constants are predefined; you can refer to them by name in *LOGiiX* scripts. You can use these predefined constants to make *LOGiiX* scripts more readable and easier to write:

PREDEFINED CONSTANT	*TYPE*	*VALUE*
True	Boolean	True
False	Boolean	False
Null	Integer	0
Obj_Null	Integer	4294967295

We recommend that authors use only OBJ_NULL in a 32-bit environment because the integer representation for OBJ_NULL is 4294967295 (0xFFFFFFFF).

The constant **Null** with its zero value can be used for all functions that return a handle to a document. The constant **Obj_Null** can be used for all functions that require or return a handle to an Object.

Variables

Variables are used to hold data temporarily in a *LOGiiX* script. You can think of the variable as a container to hold a value, much like a cell in a table or spreadsheet. Variables are always assigned by using an assignment operator (**:=**). An identifier labels the variable. For example, if you want to use a variable for employee names in a *LOGiiX* script, you could use something like **Employee_Name** to identify the variable; the value for this variable would be a particular employee's name, so the syntax for the script line would read something like **Employee_Name := "John Smith";** The value for the **Employee_Name** variable in this case is a text string, **John Smith**.

Unlike many high-level programming languages (Pascal or C, for example), *LOGiiX* is a nondeclarative language. In other words, variables may be used without first describing each variable and its contents. A variable may be assigned a value of any type at any time during the execution of a script. *LOGiiX* checks variable types when the scripts are executed. Variable type inconsistencies then show up as errors, but they aren't evident when scripts are being compiled.

Variables are used to put data into functions. For example, the **MoveWindow** function needs to know which window to move. As shown in the following script, you can use the **GetTopDocID** function to set a variable that represents the active document window. The **MoveWindow** function can use that variable to identify and move the document:

```
Begin
    hDoc := GetTopDocID ()
    MoveWindow (hDoc,0,0,150,150);
End
```

Variable values can change value as a script executes. For example, in the following script, *LOGiiX* increments the variable counter until it reaches six, and then the function terminates:

```
#LOGiiX
Function Main ()
Begin
hDoc := ButtonDoc ();
counter := 1
While (counter <>6) do
Begin
   MessageBox (counter);
   counter := counter +1;
End;
End
```

LOGiiX variables are assigned values in 'assignment' statements that take the following form:

[Identifier] := [Expression]

For example:

```
MyBoolean := true;
MyInteger := MyInteger + (A * (3 + B));
MyString := "Large house";
```

Assignment statements evaluate the expression on the right side of the **:=** operator and assign the value of that expression to the variable on the left side.

Identifiers

The name that you assign to a variable, called the identifier or ID, uniquely identifies that variable to *LOGiiX*. For example, you may decide while writing a script that you need a variable called **Result.** Every time you refer to **Result** in your script, *LOGiiX* knows that you mean the variable with that name.

You can use identifiers to designate both variables and functions. An identifier consists of a sequence of alphanumeric characters that begins with an alphabetic character; it may also contain underscores (_). Identifiers can contain any combination of upper- and lower-case characters.

Examples of valid identifiers:

a

a5

First_Name

address_line_1

Examples of invalid identifiers:

5a

_name

5a is invalid because it begins with a numeral, not a letter. The underscore at the beginning of **_name** invalidates that term as an identifier.

LOGiiX interprets **first_name** and **First_Name** as the same identifier. When you choose an identifier for a new variable, it's a good idea to use a name that reflects the variable's content. For example, call a variable **Mortgage_Amount** rather than **A** or **amt**. This makes your scripts easier to understand.

Some *LOGiiX* variables contain 'pointers' to documents or Objects (the hypertext components of a *GUIDE* document). These pointers are referred to as handles. If a variable contains a handle, it is a good idea to prefix that variable's identifier with a lowercase **h**; for example, **hDoc**.

Variable Scope

All *LOGiiX* variables can be used anywhere in a script and must therefore be given unique names. As you work with new functions, you must be careful to ensure that a variable name has not been used somewhere else in the script.

In the following example, *LOGiiX* would see the variable **Sum** used in the functions **Main** and **Make_2** as the same variable. The script writer, however, may have intended **Sum** to be unique in each case.

```
Function Main ()
Begin
    Sum := 1;
    Make_2 (Sum);
End
Function Make_2 (Sum)
Begin
    Sum := 2;
End
```

In this example, **Main** initially sets **Sum** to 1, but by the time **Make_2** executes, **Sum** has a value of 2. Even though the script writer may have intended otherwise, the variable **Sum** referred to in the functions **Main** and **Make_2** is actually the same variable.

Global Variables

You can create special variables called *global variables* in *LOGiiX*. Global variables retain their value between different executions of a *LOGiiX* script. Like normal *LOGiiX* variables, these variables can be referred to anywhere within a script. Once created, however, global variables can be used by all *LOGiiX* scripts run in a particular session. Global variables provide a way to pass data between *LOGiiX* scripts.

You declare global variables in the same way as the other variables described earlier in this user's manual, except that global variables are indicated by the word **global** before the variable name; for example:

```
Global A;
```

or

```
Global B := 3;
```

You may optionally initialize a global variable with a constant value when you create it. However, this initialization occurs only if the global variable has not already been created. For example:

```
Global A := 0;
Function Main ()
Begin
    A := A + 1;
End
```

During the first run of the this script, **A** is created and initialized at **0**, then **Main** increments it to **1**. During the second run of the script, *LOGiiX* attempts to create **A** but finds that it already exists, so it uses the value it already has. **Main** increments the variable **A** again during the next run of the script, so **A** has a value of **2** after the second run.

To use a global variable in another script, that script must also contain a **global** declaration for that variable. This informs *LOGiiX* that the script wants to create a new global variable or use the value of an existing global variable.

Global statements must appear before or within function definitions such as **Function Main ()**, not inside them. It's usually best to declare all global variables first in a script, before you write any function definitions.

Reserved Words

A number of reserved words have special meaning to the *LOGiiX* compiler. These words should not be used as variable or function names. Reserved words in *LOGiiX* are:

and	**false**	**not**	**then**
begin	**function**	**null**	**true**
do	**global**	**obj_null**	**while**
else	**if**	**or**	
end	**include**	**return**	

When interpreting reserved words, *LOGiiX* does not distinguish between upper- and lowercase letters. For example, the compiler recognizes the reserved word **While** if it appears in a script as **while**, **WHILE**, or **wHiLe**.

Limitations

Limitations in *LOGiiX* relate to the length of lines in a script, length of identifiers, and length of string constants:

- Maximum length of a line in a *LOGiiX* script:
  400 characters
- Maximum length of a *LOGiiX* identifier:
  511 characters
- Maximum length of a string constant:
  511 characters

The amount of memory a *LOGiiX* script requires varies from script to script. You should be aware that large scripts may exceed this limit and fail to run. You can sometimes overcome these limits by using **include**. Strings may be concatenated together using the plus operator (**+**).

CHAPTER 3

LOGIIX OPERATIONS AND LANGUAGE STATEMENTS

Variables in *LOGiiX* can be combined by using arithmetic and logical operators. These provide the basic addition, subtraction, multiplication, and division operations as well as arithmetic comparison and logical functions.

As an example of the use of arithmetic and comparison operators, suppose you want to add the total owed on two credit cards and, if the combined amount is less than $5000, buy a new car. The *LOGiiX* script to do this calculation is:

```
#LOGiiX
Function Main ()
Begin
    Total := (Card_A + Card_B);
    If (Total < 5000) then
    Advice := "Buy the car";
End
```

The addition of the variables **Card_A** and **Card_B** is achieved by the arithmetic operator, **+**. The result is tested using the logical operator for less than, **<**. If **Total** is less than 5000, the comparison evaluates as true and the variable **advice** is set to "**Buy the car.**"

LOGIIX OPERATORS

OPERATOR	OPERATION	OPERAND TYPE	RESULT TYPE	EXAMPLE
+	Add	Integer/String	Integer/String	A+B
-	Subtract	Integer	Integer	A-B
*	Multiply	Integer	Integer	A*B
/	Divide	Integer	Integer	A/B
%	Remainder on division	Integer	Integer	A%B
=	Equal to	Integer/String/Boolean	Boolean	A=B
<>	Not equal to	Integer/String/Boolean	Boolean	A<>B
>	Greater than	Integer/String	Boolean	A>B
<	Less than	Integer/String	Boolean	A<B
>=	Greater than or equal to	Integer/String	Boolean	A>=B
<=	Less than or equal to	Integer/String	Boolean	A<=B
and	Logical and	Boolean	Boolean	A and B
or	Logical or	Boolean	Boolean	A or B
not	Logical not	Boolean	Boolean	not A
s[i]	String item reference	s=String i=Integer	Integer	My_String[3]
:=	Assign	Any type	Operand type	A=B

Expressions

An expression in *LOGiiX* is a sequence of variable names, brackets, constants, function calls, and arithmetic and logical operators that develop a result. The following are all valid expressions:

```
(A = B) and (C <> D) or (E and F)
Y + Square (X) * 2
A and (B or C) or not (D and E)
OpenFile ("xyz.txt")
```

Expressions in *LOGiiX* are evaluated from left to right and can deliver a value of any supported type. You must, however, ensure that all the variables and constants involved in arithmetic and logical operations are the same type.

String Operators

You can reference a single character inside a string variable by the use of the string item reference operator **[ ]**. The index (character number) in a string variable is contained between square brackets.

The following script assigns one integer value to the variable **B** and another to the variable **C**. This operator may be used on either side of an assignment:

```
A := "This is a string";
B := A[8];
C := A[7];
```

The next example assigns the variable **A** with the string **this is another string**, and then reassigns the first letter in each word with its capitalized version to return **This Is Another String**:

```
A := "this is another string";
A[1] =  T ;
A[6] =  I ;
A[9] =  A ;
A[17] =  S ;
```

A letter surrounded by single quotes denotes the character's ANSI value. If you enclose characters in double quotes, they're passed along as a string value.

Strings in *LOGiiX* may be concatenated with the **+** operator. For example:

```
A := "This " + "is " + "a " + "long " + "string";
```

The above line of the script assigns the value **This is a long string** to the variable **A**.

String values may be compared for equality and alphabetic precedence with the **=**, **<>**, **>**, **<**, **>=**, and **<=** operators. For example:

```
A := "abc";
B := "zbc";
C := A > B;   {False}
D := A < B;   {True}
E := A >= B;  {False}
F := A <= B;  {True}
G := A <> B;  {True}
H := A = B;   {False}
```

Statements

LOGiiX provides five types of language statements for use in scripts:

- Assignment statements
- If statements
- While statements
- Include statements
- Return statements

Assignment Statements

In 'assignment' statements, *LOGiiX* variables are assigned values that take the following form:

```
[Identifier] := [Expression]
```

For example:

```
MyBoolean := true;
MyInteger := MyInteger + (A * (3 + B));
MyString := "Large house";
```

Assignment statements evaluate the expression on the right side of the **:=** operator and assign the value of that expression to the variable on the left side.

If Statements

In *LOGiiX,* an **if** statement conditionally executes a statement or block of script based on the value of an expression. It can take one of two forms:

```
If [expression] then [statement or block];
```

or

```
If [expression] then [statement or block] else
[statement or block];
```

In the first form, the statement executes only if the expression evaluates as true. In the second, the statement that follows the keyword **then** executes if the expression evaluates as true; if the expression is false, the statement that follows **else** executes.

An example of the first form of an **if** statement:

```
If (Day = 1) then
Begin
   Task := "Hold weekly meeting";
End;
```

An example of the second form of an **if** statement:

```
If (Day = 1) Then
Begin
   Task := "Hold weekly meeting";
End
Else
Begin
   Task := "Review major accounts";
End;
```

While Statements

The **while** statement provides a looping mechanism in *LOGiiX*. This statement takes the following form:

```
while [expression] do [statement or block];
```

The expression is evaluated and, if the result is the Boolean value **true**, the statement or block executes. The expression is evaluated again after each execution of the statement or block until it delivers the Boolean value **false**. For example:

```
A := 1;
B := 5
While (A < 15) do
Begin
   B := B * A;
   MessageBox (B);
   A := A + 1;
End;
```

Include Statements

An **include** statement inserts text from a given Definition into the script being compiled and enters the script into the Definition's text so you can include the same instructions in several scripts without rewriting code. This example shows a Definition that inserts an **include** statement in its script text; the exclamation mark separates the document's name and Object identification:

```
   Include "[document name]![Definition ID number
ornamel";
```

Because the **include** statement combines text for a given Definition in the script currently being compiled, included Definitions should not contain the **Main** function or start with the statement **#LOGiiX**. You can obtain the Definition ID or name by activating **Show Symbols** to display the symbols and IDs in the document or by using **Set Attributes** to name the Definition.

You can omit the document name in an **include** statement if the Definition you want to include resides in the same document as the Definition that contains the script with the **include** statement. For example:

```
   Include "!123";
   {This will include Definition 123 from the active
document.}
   Include "abc.gui!999";
   {This will include the Definition with the Object
ID 999 from abc.gui.}
   Include "c:docs\sample.gui!test";
   {This will include the Definition "Test" from the
document c:docs\sample.gui}
```

You can use an **include** statement to share a group of functions between a number of scripts without having to maintain multiple copies of the same text in each Button's Definition. **Include** statements may appear only before or between functions and must not appear inside them. The statements in braces ({}) are comments that describe the script but do not affect its functionality.

Return Statements

The **return** statement in *LOGiiX* stops the execution of the function that includes it and returns to the point in the script where the function was called. The syntax for a **return** statement is:

```
Return [Expression]
```

The value of the expression returns to the value that the expression had when the function was called, for example:

```
Function Square (Param)
Begin
   Return (Param * Param);
End

Function Main ()
Begin
   Result := Square (4);
End
```

In this example, the function **Square** is passed to the parameter **4** and returns the result of four times four. After the return statement is executed, the variable **Result** has a value of 16.

Comments

It's sometimes useful to be able to place text within a script that won't be interpreted. You can add these kind of 'comments' to a *LOGiiX* script by enclosing text you don't want *LOGiiX* to execute in braces ({}), for example:

```
#LOGiiX
{This script adds two variables and prints the
result in a message box.}
Function Main ()
Begin
    A := 1;
    {Make the first variable equal one.}
    B := 2;
    {Make the second variable equal two.}
    C := A + B;
    {Add the two variables.}
    MessageBox ("A = 1, B = 2, A + B = ", C);
    {Print the result.}
End
```

Functions

LOGiiX scripts can be split up into logically separate units called 'functions'. As you've already seen, all *LOGiiX* scripts contain at least one function, **Main**.

LOGiiX provides a set of function calls to perform actions in *GUIDE Author, GUIDE Reader,* and *GUIDE Viewer,* which are referred to as 'callback functions'. You define these callbacks by combining several functions together to expedite your work. You may also define any number of functions for your own use. Your personal functions can have any name that is not a reserved word (please see "Reserved Words" in Chapter 2, *"LOGiiX* Fundamentals," earlier in this manual).

With the exception of **Main**, all *LOGiiX* functions can have parameters. In this example, the parameters **X** and **Y** are passed to the function **Multiply**. They are multiplied and the result is assigned to **Z**. The value of **Z** is then returned as the result of the function **Multiply**. Note that the variables **X** and **Y** contain only copies of the actual parameter values passed when the **Multiply** function is called.

```
Function Multiply (X, Y)
Begin
   Z := X * Y;
   Return Z;
End
```

A function call to **Multiply** could look like this:

```
A := Multiply (2, 4);
```

Both the parameters passed to a function and their return values may be of any type; however, an error occurs if a parameter is not the type *LOGiiX* needs to execute the function. The following example attempts to multiply a number by a string and this operation cannot be evaluated. An error message displays when this script attempts to execute.

```
A := Multiply (2, "House");
```

There's no limit to the number of parameters that can be passed to a function; however, the function **Main** cannot have any parameters. Unexpected results also occur if a function is called with the wrong number of parameters. If a *LOGiiX* function is called with too few or too many parameters, it won't execute and will return an error.

Special LOGiiX Definitions

Generally, *LOGiiX* scripts in a *GUIDE* document's Definitions window execute when a reader clicks on a Command Button. You can, however, ask *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer* to execute a *LOGiiX* script automatically in certain circumstances if you assign **%on_open**, **%on_move**, **%on_close**, **%on_activate**, **%on_deactivate**, or **%on_frame** as the Object name for the Definition that contains the script:

%on_open	Executes when the *GUIDE* document that contains the script first opens
%on_move	Executes when the document is moved or resized
%on_close	Executes just before the document closes
%on_activate	Executes when the document is made active and brought to the front of all open *GUIDE* documents
%on_deactivate	Executes when the *GUIDE* document that contains the script becomes inactive; that is, another document becomes active.
%on_frame	Executes whenever the Frame in the *GUIDE* document that contains this item changes.

NOTE:

The Object names for these special Definitions must be typed in lower case letters in GUIDE Author's Object Attributes dialog to be properly recognized by GUIDE Author, GUIDE Reader, or GUIDE Viewer.

If you use one of these expressions in a Definition's Object name, the script that Definition contains automatically executes when the event referenced in the Object name takes place. For example, if you assign **%on_open** as a Definition's Object name, the script in that Definition executes every time the *GUIDE* document associated with that Definition opens.

You can use these special Object names only once per document. If you assign a special event name to more than one Definition in a *GUIDE* document, only the first script executes. You can, however, combine scripts into one event-specific Definition.

These special event Definitions are most useful to execute *LOGiiX* scripts without Command Buttons. For example, you can write a script that asks readers for a password as soon as the document opens. If the password isn't correct, the document closes.

Dynamic Data Exchange (DDE)

In addition to scripting language functionality, *LOGiiX* provides a Dynamic Data Exchange (DDE) interface that allows you to link *GUIDE* documents with other Windows applications. In a DDE environment, an application called the 'server' performs tasks for other applications referred to as 'clients'.

For example, a reader opens a *GUIDE* document that contains a sales report from an operating division. The document's Definitions window contains a *LOGiiX* script designed to execute automatically when the document opens. The script sets up a DDE link to contact Microsoft Excel and requests the latest monthly sales figures from an Microsoft Excel spreadsheet. Microsoft Excel finds the figures and sends them to *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer.*

NOTE:

To learn more about DDE, please see the documentation provided with the Windows Software Developer's Kit.

In the following example, either *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer* acts as the client — through *LOGiiX* — and asks Microsoft Excel to perform a task on its behalf. In its role as the server, Microsoft Excel performs the task and returns some data via the DDE link. This is only one example of the possible types of interaction that can take place between Windows applications that have a DDE interface. Other examples are discussed in greater detail later in this manual.

LOGiiX as a DDE Server

DDE implementation in *LOGiiX* provides the interface to allow *GUIDE Author, GUIDE Reader,* and *GUIDE Viewer* to act as both DDE clients and DDE servers. *LOGiiX* responds as a DDE server when another application (the client) tries to initiate a DDE conversation with it. When *LOGiiX* responds as a DDE server, *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer* perform tasks on behalf of and at the request of another application.

Initiating a DDE Link

You can use the **DDEInitiate** function to initiate a DDE link with *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer* from other Windows applications that support a DDE interface. All applications that support DDE respond to three arguments:

- Application name
- Topic
- Item

To open a channel to *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer,* use the text value "GUIDE" in the application name argument of the **DDEInitiate** routine. If *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer* is running, it responds to the **Initiate** message.

GUIDE Author, GUIDE Reader, and *GUIDE Viewer* respond to the topic "system" for DDE system information and to the *GUIDE* document's file name. If you want to initiate a DDE link to a specific document, supply the document name as the topic (**saleinfo.gui** or **c:\trucks\docs\saleinfo.gui**, for example).

NOTE:

The preload flag should be set for LOGiiX in the initialization file (infacces.ini) if you want GUIDE Author, GUIDE Reader, or GUIDE Viewer to respond to the DDEInitiate message. LOGiiX connects GUIDE publications and other applications through DDE. For more information about the initialization file, please refer to "Getting Started" in Welcome to GUIDE Author.

An item is data within a topic. If you use the "system" topic, *LOGiiX* responds to the following items:

ITEM	*DESCRIPTION*
SysItems	A list of all other items you can use with the "system" topic (topics, status, formats, selection)
Topics	A list of all available topics (open *GUIDE* documents)
Status	The status of the *LOGiiX* DDE interface; if *LOGiiX* is ready to receive commands, it responds with 'Ready'; otherwise, it responds with 'Busy'
Formats	A list of data formats *LOGiiX* supports
Selection	Returns anything selected in the *GUIDE* document in the format **Topic!Item**

The **Item** returned is the name of the Object that contains the selection, if it has a name; otherwise, it's the Object ID. Conversations initiated with specific *GUIDE* documents enable the client application to send ('poke') data into or request data out of named or numbered Objects in a *GUIDE* document. In such a request, the **Item** string should be the number or name of the *GUIDE* Object. In a 'poke' operation, data sent from the client replaces the contents of the *GUIDE* Object and a request returns the entire contents of the *GUIDE* Object. When a client no longer needs to talk to *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer* over the DDE link, the client notifies *LOGiiX* through the **DDETerminate** function.

Using DDE Execute

LOGiiX may receive DDE execute messages that allow other applications to execute *LOGiiX* functions remotely. The text sent to *LOGiiX* consists of several *LOGiiX* function calls, each contained between a matching pair of square brackets, for example:

```
   [NameOpen ("test.gui")] [NameAction
("test.gui","123")]
   [NameClose ("test.gui")]
```

This sequence of execute statements would be translated by *LOGiiX* into the following script, which then executes in the usual way:

```
Function Main ()
Begin
   NameOpen ("test.gui");
   NameAction ("test.gui", "123");
   NameClose ("test.gui");
End
```

Only simple *LOGiiX* scripts may be transferred by these means; an extension exists so that an arbitrary script may be sent across a DDE link. This requires that the first execute statement contains **#LOGiiX** to call the *LOGiiX* interpreter. All subsequent execute statements should be standard lines of *LOGiiX* script complete with the semi colons in the appropriate places; for example:

```
[#LOGiiX] [Include "!Button26";] [Global A := false;]
[Function Main ()] [Begin] [NameOpen ("test.gui");]
[A := NameAction ("test.gui", "123");]
[if (A = true) then] [Success ();]
[NameClose ("test.gui");] [End]
```

This sequence of execute statements would be translated by *LOGiiX* into the following scripts, and then executed in the usual way:

```
Include "!Button26"
Global A := false;
Function Main ()
Begin
   NameOpen ("test.gui");
   A := NameAction ("test.gui", "123");
   If (A = true) then
      Success ();
   NameClose ("test.gui");
End
```

LOGiiX as a DDE Client

A set of functions enable *LOGiiX* to act as a client in DDE conversations with other applications. When *LOGiiX* acts as a DDE client, it sends DDE messages to another application, asking it to perform tasks on behalf of *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer*. The DDE functions are:

FUNCTION	*PURPOSE*
DDEInitiate (Application, Topic)	Initiate a DDE link with another application
DDETerminate (Channel)	Terminate a DDE link with another application
DDEPoke (Channel, Item, Data)	Poke data into another application
DDERequest (Channel, Item)	Request a data item from another application
DDEExecute (Channel, Data)	Send data to another application for execution

DDE Functions Example

The following example illustrates use of the DDE functions. The script shown:

- Opens a DDE link with Microsoft Excel and the spreadsheet document **Sheet1**
- Pokes sample data 'John Smith' into the cell **R1C1** in **Sheet1**
- Requests the data back again
- Uses the **Beep** command to tell Microsoft Excel to make a sound to indicate the end of the DDE conversation
- Prints the result of the 'request' operation in a message box

```
Function Main ()
Begin
Channel := DDEInitiate ("Excel", "Sheet1");
{Initiate a DDE link with Microsoft Excel}
If (Channel = 0) then
{Check that the link is OK}
    Begin
    MessageBox ("Error initiating DDE link");
    End
Else
    Begin
        {Store a name in the spreadsheet}
        DDEPoke (Channel, "R1C1", "John Smith");
        {Get the name back}
        Employee_Name := DDERequest (Channel, "R1C1");
        {Ask Microsoft Excel to beep}
        DDEExecute (Channel, "{Beep()}";
        {Close the DDE link}
        DDETerminate (Channel);
        MessageBox ("Employee = ", Employee_Name);
    End;
End
```

You can use the **Declare** and **Call** functions together to access functions outside *GUIDE Author, GUIDE Reader,* and *GUIDE Viewer,* such as dynamic link libraries (DLLs). For more information, please see **Declare** and **Call** in the *LOGiiX Command Reference* provided online as part of *GUIDE Author's* help system.

CHAPTER 4

USING LOGIIX WITH MULTIMEDIA

Using multimedia in *GUIDE* documents isn't complicated. Start by deciding which medium is most appropriate, given your information and audience. For example, consider this scenario: You would like to distribute a *GUIDE* publication that teaches people to tune an automobile engine. One of the procedures is changing the spark plugs.Rather than explain it with words, you can link video sequences to the document with Buttons and include a full motion demonstration of a person changing spark plugs. Readers who want to see how it's done can click on a Button and view the video sequence; those who already know how to change spark plugs can skip the demonstration.

Consider another scenario: Your *GUIDE* document is a language lesson. Users read phrases on the screen and click on the phrases to hear how they are spoken. The phrase texts are *GUIDE* Command Buttons linked to digital audio recordings.

Any kind of medium that can be digitized can be incorporated into *GUIDE* publications. For example, there are services that create video discs from video tapes. By now, most people are familiar with audio compact discs (CDs); audio from CDs can be incorporated into a computer application. MIDI (Musical Instrument Digital Interface) gives you the capability to control your own musical instrument (for example, a synthesizer) from your computer.

Incorporating multimedia in *GUIDE* documents requires careful planning and some simple programming with *LOGiiX*. You'll find the results are well worth the minimal effort involved. Opening your publications to media besides text and graphics adds new depth and dimension to the information you're presenting.

Three *LOGiiX* functions—**mciSendString**, **mciGetLastError**, and **mciGetErrorString**—enable you to embed animation, digital audio, and music into *GUIDE* documents. These functions give you access to the capabilities of the Windows Media Control Interface (MCI). The MCI allows you to control media through a high-level interface and to control video displays, audio hardware, video disc players, and videotape players.

Device Types

When you use a media file in a script, you specify what type of file it is; this tells the MCI what kind of device to use. The types are:

TYPE	*DESCRIPTION*
avivideo	Audio/video interleaved format (Windows standard for synchronized video and sound)
mmmovie	Movie (animation) player
cdaudio	CD audio player
dat	Digital audio tape player
digitalvideo	Digital video (video in a window but not GDI-based)
other	Undefined device
overlay	Overlay device (analog video in a window)
scanner	Image scanner
sequencer	MIDI sequencer
vcr	Videotape recorder or player
videodisc	Videodisc player
waveaudio	Audio device that plays digitized waveform files

The device types installed in a system are listed in the **[MCI]** section of the **system.ini** file. Please see your Windows or Microsoft Multimedia Development Kit documentation for more information about the **system.ini** file and device types.

Using LOGiiX Multimedia Functions

Windows includes a file named **mmsystem.dll**; this dynamic link library provides functions that enable *GUIDE* documents to display multimedia files and control various devices. The **mmsystem** DLL uses the MCI to translate between *GUIDE Author, GUIDE Reader,* or *GUIDE Viewer* and multimedia components.

Three *LOGiiX* functions—**mciSendString**, **mciGetLastError**, and **mciGetErrorString**—pass text strings that instruct the MCI to play media elements. You can use a fourth function, **mciWaitNotify**, with **mciSendString** to execute another command while the media is playing. You can also use the *LOGiiX* multimedia functions to test for a procedure's success and to display any error messages generated. For more information about MCI functions, please see the documentation supplied with Windows or the Microsoft Multimedia Development Kit.

To use *LOGiiX* multimedia functions, just include them in scripts. For convenience, you can set up a standard Definition that contains a routine with these functions, and then refer to that Definition in subsequent scripts. You may also want to make the Definition a Glossary item so you can use it later.

Using mciSendString

The correct syntax for this function is:

```
mciSendString (mciString);
```

This function accepts a command in the form of a character string that controls internal multimedia elements (animation and sampled sound, for example) and external elements including video, CD, and audio. This *LOGiiX* function calls the **mmsystem** (multimedia system) **mciSendString** function with the command string specified in the **mciString** parameter.

An optional second parameter can be added to this string (see **mciWaitNotify**):

```
mciSendString (mciString, Value);
```

Each string can be as long as 511 characters. If you need to use more characters, you can concatenate strings with a **+** operator (please see "String Operators" earlier in this manual for additional information).

Using mciWaitNotify

The syntax for this function is:

```
mciWaitNotify ()
```

This function has no parameters; it's used only with **mciSendString**. It allows you to execute other commands and actions before you close the file. For example, if you're running a video, you could use the optional parameter in **MciSendString** (with the value **2** for **Notify**) to recognize **mciWaitNotify** and allow the execution of another action while the video is still running.

Using mciGetLastError

The syntax for this function is:

```
mciGetLastError ();
```

This function returns the error value stored by the last call to the **mciSendString** function. It returns zero if the last call was successful or if no call has been made to **mciSendString**. Otherwise, it returns one of the error codes defined by the MCI. Please refer to the documentation provided with the Microsoft Multimedia Development Kit for a list of error codes defined by the MCI.

Using mciGetErrorString

The syntax for this function is:

```
mciGetErrorString (LastError);
```

This function displays the error message associated with the code returned by **mciGetLastError**. For example, if **mciGetLastError** returns 10, **mciGetErrorString** displays the "Device not ready" message. If the call to the **mmsystem mciGetErrorString** function fails to return an error string, **mciGetErrorString** returns a default error code.

Using Audio

The sample script shown below plays a waveform file named **hello.wav**. You can define an 'alias' for a file name in a script so that you don't have to specify the full name of the file each time you want to use it. Comments in braces ({}) explain each process. This script uses the alias **sounds** for the **hello.wav** file.

This script also represents typical *LOGiiX* multimedia programming. No matter which medium you use—sound, video, etc.—your script should follow the basic structure shown in this example:

```
#LOGiiX
Function Main ()
Begin
    {Identify waveform file to play.}
    File := "c:\\GUIDE\\hello.wav";
    {Send command to open waveform file to the MCI.}
    mciSendString ("Open " + File +  " Type waveaudio
alias sounds");
    {Get the error condition of the MCI open.}
    Error := mciGetLastError ();
    If (Error = 0) then
    {If everything is OK, play the waveform file.}
    Begin
        mciSendString ("Play sounds from 1 wait");
        mciSendString ("Close sounds");
    End
    Else
    {If an error message resulted from the open
command, display the error message and return.}
    Begin
        Errmsg := mciGetErrorString (Err);
        MessageBox (Errmsg);
    End;
End
```

Using Video

The next sample script displays a video sequence. It opens the videodisc player, plays a sequence from Frame 100 to Frame 1000, and then pauses. In this case, the device alias is **vdisc**.

```
#LOGiiX
Function Main ()
Begin
    {Send command to select video disc player.}
    mciSendString ("Open type videodisc alias
vdisc");
    {Get the error condition of the MCI open.}
    Err := mciGetLastError ();
    If (Err = 0) then
    {If everything is OK, play the video.}
    Begin
        mciSendString ("Play vdisc from 100 to
1000 wait");
        mciSendString ("Close vdisc");
    End
    Else
    {If an error message resulted, display the
error message and return.}
    Begin
        mciGetErrorString (Err);
        MessageBox (Err);
    End;
End
```

An Advanced Technique

This section describes how to write a *LOGiiX* library function to make MCI programming easier. You can set up a standard Definition that includes a routine with the new functions, and then refer to that Definition in subsequent *LOGiiX* scripts. You might also want to include the Definition as a Glossary entry for later use.

For example, you can create a *LOGiiX* library function **mmcmd** as shown in the following script; **mmcmd** combines the steps of sending a string to the MCI, displaying any string in a message box, and returning the error string. When you write a script of your own following this example, remember to assign an Object name for the **mmcmd** Definition in *GUIDE Author's* **Object Attributes** dialog.

The **mmcmd** Definition is:

```
    Function mmcmd (String)
    Begin
        {Pass the string to the MCI.}
        Ret := mciSendString (String);
        {Retrieve the error number.}
        Error := mciGetLastError ();
        If (Error <> 0) then
        {If there is an MCI error, display the proper
error string to the user.}
        Begin
            Errmsg := mciGetErrorString (Error);
            MessageBox (Errmsg);
        End;
        {Function returns the MCI error number.}
        Return (Error);
    End
```

The next script shows how to include and use the **mmcmd** function to play the waveform file described earlier in "Using Audio":

```
    #LOGiiX
    Include "!mmcmd";
    Function Main ()
    Begin
        File := "c:\\GUIDE\\hello.wav";
        Err := mmcmd ("Open" + File + " Type waveaudio
alias sounds");
        If (Err = 0) then
        Begin
            mmcmd ("Play sounds from 1 wait");
            mmcmd ("Close sounds");
        End;
    End
```

Troubleshooting

If you get unexpected results from a script or media player, you should check to make sure that:

- All external hardware (the video disc player, amplifier, or any other equipment) has power.
- All cables and connections between the computer and external equipment are secure.
- You installed all required software drivers correctly.
- The *LOGiiX* scripts that correspond to the various tasks are correct. If there is a syntax error, you should see an error message that identifies the problem; however, if the script contains logic errors (for example, two lines are in the wrong order), you won't see an error message. Check the script.
- You're running in the correct Windows mode.

Also remember to consult the documentation for both the equipment and software you are using with your *GUIDE* publication.

CHAPTER 5

LOGIIX ERROR MESSAGES

LOGiiX usually attempts to report errors.Error messages with an error number, line number, and the approximate position of the error in the line appear on the screen when the script executes. The error number can help you debug a *LOGiiX* script.

This chapter presents a brief summary of *LOGiiX* error messages with their error numbers and what they mean. The error number and message are shown in bold; the description follows the message.

ERROR NO. *ERROR MESSAGE*

1 Expecting a FUNCTION, GLOBAL or INCLUDE

A function Definition, global declaration, or Definition **Include** statement was expected but not received. Check to see if you have a surplus **End** statement in the middle of a function or if you've left the braces (**{ }**) off a comment. Check the syntax of your entry.

2 Expected an integer, Boolean, or string value

Something other than an integer, Boolean, or string was used when initializing a global variable (an identifier, perhaps). The initial value for a global variable must be a constant, integer, Boolean value, or string. Check to see that you are using one of these as the initializer.

3 This function has already been defined

An attempt has been made to define a function with the same name. Check to see that you haven't used a Definition with **Function Main ( )** in an 'include' statement. This message may also display when another function name has already been used in an included definition.

4 Expecting a primary expression

LOGiiX expected an expression but found something else. Primary expressions are expressions that include **if**, **while**, or **begin**, a function call, or an assignment. This message appears if you have a surplus semicolon or if you try to embed a function block inside another.

5 Insufficient memory

This script exceeds the limit permitted by *LOGiiX.* The limit varies according to the script or possibly because of the memory Windows is currently using.

6 Abandoned by operator

The user pressed **Esc** to halt execution of a *LOGiiX* script.

7 Call to nonprocedure

This error occurs when you attempt to use a variable name as if it were a function. Please see Error #14.

8 Function called with too few parameters

A function has been called without enough parameters. Check the syntax of the function and the parameter list.

9 String reference out of bounds

LOGiiX is trying to access a character that's outside the bounds of a string. This message appears when you use the string item reference operator or to try to access a character before or after the data in the string.

10 Attempting to reference a nonstring as a string

An attempt has been made to access a nonstring variable as a string. Check the type of data assigned to the variable. Either it was never initialized or Boolean or integer data has been assigned to this variable somewhere else in the script.

11 Divide by zero error

A script has attempted to divide by zero.

12 Wrong type of parameter

An arithmetic or logical operation has been attempted with data of the wrong type for the operation. For example, this message appears when you try to subtract two strings.

13 Stack overflow

This script is too complex for *LOGiiX* to handle. This error also occurs during a recursive function call (a function that calls itself).

14 Undefined function

A function name has been used for a function that hasn't been declared. This error occurs if the function **Main** has been omitted from a script or if you misspell the name. Check the function.

15 String is too long to display

The total length of the string produced by concatenating **Message** parameters in **Ask**, **Answer**, or **MessageBox** functions cannot exceed 511 characters.

16 Obsolete

17 Negative indexing into strings is not allowed

An attempt has been made to access a character in a string with a negative index. This message appears when you use the **Substring** function and try to address characters before the start of the string. The first character in the string is **1** (one).

18 Too many nested levels of INCLUDE

There's no fixed limit on the number of **include** statements you can use in a single script. Generally, however, you shouldn't attempt to nest **include** statements past nine levels.

19 INCLUDE statement must specify a Definition ID

An **include** statement doesn't have a necessary Definition identifier. Check the script to see if the identifier ID is missing.

20 INCLUDE fails to open document

An attempt has been made to include a Definition from a *GUIDE* document that either does not exist or could not be opened. This error message can appear, for example, if you try to include something from the active *GUIDE* document when the document has not yet been saved.

21 INCLUDE can only include other Definitions

An attempt has been made to include a *GUIDE* Object other than a Definition. This occurs if the Definition ID identifies an Object that is not a Definition or if the Definition ID doesn't identify any Object in the specified document.

22 INCLUDE statement must be of the form [docname]![Definition ID]

A syntax error has occurred in an **Include** statement. Check the syntax of the statement.

23 Invalid integer constant

An invalid integer constant has been entered. This occurs only when *LOGiiX* finds the end of the Definition while it's parsing an integer constant.Turn on **Show Symbols** and check to make sure you've entered the script correctly inside the Definition symbols.

24 Invalid hexadecimal constant

An invalid hexadecimal constant has been entered. This occurs if the first part of a numeric constant is missing or invalid.

25 String too long

A string constant has too many characters. The maximum length of a string constant is 511 characters. Check to ensure that you enclosed the string in quotation marks.

26 Empty character constant

LOGiiX has found a character constant of the form **' '** (two single quotes with no characters between them), which is not allowed. This error also occurs if *LOGiiX* comes to the end of the Definition after it processes the opening single quote. Open the *GUIDE* document and its Definitions window in *GUIDE Author,* turn on **Show Symbols**, and check that the script you entered is inside the Definition symbols.

27 Invalid character constant

An invalid character constant has been entered. This generally occurs if you try to enter more than one character between quotes in a character constant.

28 Identifier too long

A variable or function has been declared with a name that is too long for *LOGiiX* to process. The maximum length of a variable name is 511 characters.

29 Integer overflow error

A variable has been assigned a value outside the available range (hex **0x7FFFFFFF** or decimal **2147483647**).

30 Line too long

This line of the script is longer than 400 characters, the maximum length for a single line in a *LOGiiX* script.

31 Syntax error, unexpected symbol.

LOGiiX failed to find a legal symbol at the section of the script currently being parsed. Generally, this is caused by a missing keyword or by a quote or semicolon on the line before the one identified by the error message.

32 Function called with too many parameters.

A *LOGiiX* function has been called with more parameters than it should have.

33 Function too long

This message may appear when LOGiiX attempts to execute long or complex scripts. *LOGiiX* code is compiled to a pseudo-machine code that uses a one-byte value to address function parameters, intermediate results, and constants. When the number of these 'literals' gets too big, *LOGiiX* reports this error. Try to simplify the script or break it down into multiple scripts by using **include** statements.

34 Exception handled in Call function

The function being called by the Call function threw an exception.

35 Stack corruption in Call function

Caused by the function being called failing or invalid parameters being passed to it.

— Too many nested or chained callbacks.

This error message may be detected by *GUIDE Reader* as the by-product of the execution of a *LOGiiX* script. It may occur, for example, if you have a **doc1** that has a **%on_open** script, which opens **doc2** that has its own **%on_open** script, which opens **doc3** that has... You may have too many chained callbacks for *LOGiiX* to handle. You may receive the same error message in the case of a Definition that executes a Definition that executes a Definition that... Try to simplify your scripts and run them again. This error message hasn't been assigned a number.

INDEX

Symbols

#LOGiiX command 9
%on_activate 32
%on_deactivate 32
%on_frame 32
%on_move 32
%on_open 32

B

Blocks, LOGiiX script 10
Boolean values 12

D

DDE functions
  DDEExecute 37
  DDEInitiate 37
  DDEPoke 37
  DDERequest 37
  DDETerminate 37
  Examples 38
DDE links
  Formats 35
  Selection 35
  Status 35
  SysItems 35
  Topics 35
DDE server 33
DDEInitiate Function 34
Definitions 32

G

Global variables 18

L

LOGiiX
  About 5
  Advanced technique 45
  Assignment statements 15
  Compiler 7
  DDE Client 37
  DDE Server 33
  Errors 49
  Functions
    Parameter 31
  Limitations 20
  Multimedia functions 41
    mciGetErrorString 43
    mciGetLastError 43
    mciSendString 41, 42
  Numeric values 12
  Operators 22
  Reserved words 19
  Special Definitions 32

LOGiiX data types
Boolean values 12
Numeric values 12
Strings
Backslash character 12
New line 12
Tab character 12
Strings (text characters) 12
LOGiiX functions, DDE Link 34
LOGiiX scripts 6
#LOGiiX command 9
Begin and End 10
Blocks 10
Body 10
Constants 13
Expression 23
Functions 30
Global variables 19
Hexadecimal constants 13
Interrupting 10
Key syntax 9
Operators 22
Predefined constants
false 13
null 13
True 13
Semicolon 10
Special definitions 32
Variables 14
LOGiiX statements
If 26
Include 27
Return 29
While 27
logiix.dll 7

M

MIDI (Musical Instrument Digital Interface) 39
Multimedia 39
Multimedia functions, MMSYSTEM 41

S

Sample LOGiiX scripts
About 7
Audio 44
Basic 9
Buy the car 21
DDE Execute 36
DDE Functions 38
If 26
Include 28
MoveWindow 14
Multimedia 46
Return 29
Video 45
While 27
Strings
Backslash character 12
Data type 12
New line 12
Tab character 12

T

Troubleshooting, LOGiiX 49

V

Variables 14
Assignment statements 15
Data type errors 14
Globals 18
Handles 16
Indentifiers 16